Portfolio Report 6: OpenMP

Jake Spiteri

2020

Intro to OpenMP

We first create a simple C++ program which uses OpenMP. The following file will be called `hello_openmp.cpp`.

```
#include <iostream>

int main(int argc, const char **argv)
{
    #pragma omp parallel
    {
        std::cout << "Hello OpenMP!\n";
    }

    return 0;
}
```

We compile the above code using the command

```
g++ -fopenmp hello_openmp.cpp -o hello_openmp
```

The code can then be run using

```
./hello_openmp
```

```
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
```

We see that the command in the C++ function within the block beginning with `#pragma omp parallel {}` is ran on each core — eight times!

We can manually set the number of threads to run in parallel using

```
export OMP_NUM_THREADS=4
./hello_openmp
```

```
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
```

The line `Hello OpenMP!` is now repeated four times as the program was split into four threads, and each thread printed the line `Hello OpenMP!` once.

Directives (Pragmas)

A normal program works by running one line of code at a time starting from the main function and working downwards. The single thread of execution is known as the `main` thread, which is often the only thread executed in a standard program. The program used above `hello_openmp` also has a single `main` thread, which is then split into multiple threads within the OpenMP parallel section. Each thread runs all of the code within the parallel section.

We can explicitly see this if we compile and run the following program which tells each parallel thread to print its thread number. We save the following C++ code as `hello_threads.cpp`.

```
#include <iostream>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_num_threads() 0
    #define omp_get_thread_num() 0
#endif

int main(int argc, const char **argv)
{
    std::cout << "I am the main thread.\n";

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int thread_id = omp_get_thread_num();

        std::cout << "Hello. I am thread " << thread_id
                  << " out of a team of " << nthreads
                  << std::endl;
    }

    std::cout << "Here I am, back to the main thread.\n";

    return 0;
}
```

```
g++ -fopenmp hello_threads.cpp -o hello_threads
./hello_threads
```

```
## I am the main thread.
## Hello. I am thread Hello. I am thread Hello. I am thread Hello. I am thread Hello. I am thread Hello
##
##
##  out of a team of 8
## 1 out of a team of 8
## 0 out of a team of 8
## Hello. I am thread 5 out of a team of 8Hello. I am thread
## 3 out of a team of 8
## Here I am, back to the main thread.
```

We see that the output doesn't print as we might expect, as all threads are attempting to print at the same time.

The above code utilizes some new OpenMP functions:

- `omp_get_num_threads()` — returns the number of threads in the OpenMP thread team.
- `omp_get_thread_num()` — returns the number of the thread in the team.

These functions are contained within the OpenMP header file which we import using an `#ifdef _OPENMP` guard.

There are several OpenMP directives which we may use

- `parallel` — creates a block of code which runs in parallel as it is executed by a team of threads
- `sections` — specifies different sections of code which can be ran in parallel by different threads
- `for` — used to specify loops where different iterations of the loops can be ran at the same time by different threads
- `critical` — used to specify a block of code that can only be ran by one thread at a time
- `reduction` — used to combine multiple results into a single result

We can easily add directives to our C++ code using the line

```
#pragma omp name_of_directive
```

Sections

```
#include <iostream>
#include <unistd.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

void times_table(int n)
{
    int thread_id = omp_get_thread_num();

    for (int i=1; i<=n; ++i)
    {
        int i_times_n = i * n;
        std::cout << "Thread " << thread_id << " says " << i
                  << " times " << n << " equals " << i_times_n << std::endl;
        sleep(1);
    }
}

void countdown()
{
    int thread_id = omp_get_thread_num();

    for (int i=10; i>=1; --i)
    {
```

```
        std::cout << "Thread " << thread_id << " says " << i << "...\n";
        sleep(1);
    }

    std::cout << "Thread " << thread_id << " says \"Lift off!\"\n";
}

void long_loop()
{
    double sum = 0;

    int thread_id = omp_get_thread_num();

    for (int i=1; i<=10; ++i)
    {
        sum += (i*i);
        sleep(1);
    }

    std::cout << "Thread " << thread_id << " says the sum of the long loop is "
              << sum << std::endl;
}

int main(int argc, const char **argv)
{
    std::cout << "This is the main thread.\n";

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                times_table(12);
            }
            #pragma omp section
            {
                countdown();
            }
            #pragma omp section
            {
                long_loop();
            }
        }
    }

    std::cout << "Back to the main thread. Goodbye!\n";

    return 0;
}
```

```
g++ -fopenmp omp_sections.cpp -o omp_sections
```

```
./omp_sections
```

```
## This is the main thread.
## Thread Thread 1 says 510...
##  says 1 times 12 equals 12
## Thread 1 says 9...
## Thread 5 says 2 times 12 equals 24
## Thread Thread 1 says 8...
## 5 says 3 times 12 equals 36
## Thread 1 says 7...
## Thread 5 says 4 times 12 equals 48
## Thread Thread 51 says 5 times 12 says  equals 6...
## 60
## Thread 5 says 6 times 12 equals 72
## Thread 1 says 5...
## Thread 5 says 7 times 12 equals 84
## Thread 1 says 4...
## Thread 5 says 8 times 12 equals 96
## Thread 1 says 3...
## Thread Thread 51 says 9 times  says 122 equals ...
## 108
## Thread 5 says 10 times 12 equals 120
## Thread 1 says 1...
## Thread 6 says the sum of the long loop is 385
## Thread 5 says 11 times 12 equals 132
## Thread 1 says "Lift off!"
## Thread 5 says 12 times 12 equals 144
## Back to the main thread. Goodbye!
```

By specifying `#pragma omp section` we are placing each function (`times_table`, `countdown`, `long_loop`) within its own OpenMP section. This tells the compiler that these three sections can be ran in parallel and one thread must be assigned to each section. In this case we have three sections and four threads, and so one thread will not do anything. If there are more sections than threads then the sections will be queued until there is a free thread to run it. There is no guarantee as to the order in which the threads will be run. parallelizing code over sections is not necessarily the best approach to implementing OpenMP. We can achieve better performance by parallelizing over loops.

Loops

We can use the directive `for` to parallelize over loops. This means that if we have a loop with 1000 iterations and two threads, one thread will run 500 of the iterations and the other will run the other 500 at the same time. This will provide a speed up of 2 times. Of course if we had more threads then the number of iterations would be split between the threads. If we have more threads than iterations then some threads will sit idle whilst others run the code.

It's important to note that we require each iteration of our loop to be independent, as there is no guarantee that one iteration will be ran before another.

We will save the following code as `loops.cpp`, compile it and run it, in order to demonstrate `loops` with OpenMP.

```
#include <iostream>

#ifdef _OPENMP
    #include <omp.h>
```

```
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, const char **argv)
{
    #pragma omp parallel
    {
        int nloops = 0;

        #pragma omp for
        for (int i=0; i<1000; ++i)
        {
            ++nloops;
        }

        int thread_id = omp_get_thread_num();

        std::cout << "Thread " << thread_id << " performed "
                  << nloops << " iterations of the loop.\n";
    }

    return 0;
}
```

```
g++ -fopenmp loops.cpp -o loops
```

```
./loops
```

```
## Thread Thread 0Thread  performed 125 iterations of the loop.
## Thread 7 performed 125 iterations of the loop.
## 5 performed 125 iterations of the loop.
## Thread 4Thread  performed 125 iterations of the loop.
## 1 performed 125 iterations of the loop.
## 6 performed 125 iterations of the loop.
## Thread Thread 32 performed 125 iterations of the loop.
##  performed 125 iterations of the loop.
```

```
export OPT_NUM_THREADS=4
./loops
```

```
## Thread 6 performed 125 iterations of the loop.
## Thread 1 performed 125 iterations of the loop.
## Thread 0 performed 125 iterations of the loop.
## Thread 7 performed 125 iterations of the loop.
## Thread 5 performed 125 iterations of the loop.
## Thread 2 performed 125 iterations of the loop.
## Thread 3 performed 125 iterations of the loop.
## Thread 4 performed 125 iterations of the loop.
```

Critical code

So far we have looked at parallelizing over loops and sections, but we haven't seen how to make threads work together or share information. The methods we have seen so far execute different iterations of loops over different threads, and execute different sections on different threads, but they do not share

information/variables between themselves — each thread has its own scope containing its own private variables. In order to make these variables globally accessible we must use a critical section.

Below we illustrate the use of `OpenMP`'s `critical` section by coding a Monte Carlo algorithm to approximate π.

```
#include <cmath>
#include <cstdlib>
#include <iostream>

double rand_one()
{
    return std::rand() / (RAND_MAX + 1.0);
}

int main(int argc, const char **argv)
{
        int n_inside = 0;
        int n_outside = 0;

        #pragma omp parallel
        {
                int par_n_inside = 0;
                int par_n_outside = 0;

                #pragma omp for
                for(int i=0; i<1000000; ++i)
                {
                        double x = (2 * rand_one()) - 1;
                        double y = (2 * rand_one()) - 1;
                        double r = sqrt(x*x + y*y);

                        if(r < 1.0)
                        {
                                ++par_n_inside;
                        }
                        else
                        {
                                ++par_n_outside;
                        }
                }

                #pragma omp critical
                {
                        n_inside += par_n_inside;
                        n_outside += par_n_outside;
                }
        }

        double pi = (4.0 * n_inside) / (n_inside + n_outside);

        std::cout << "The estimated value of pi is " << pi << std::endl;

        return 0;
}
```

We see that within the `omp parallel` directive, we have a `omp for` directive which allows multiple threads to work on different iterations of the for loop whilst locally storing their private variables as `par_n_inside` and `par_n_outside`. In order to combine these private variables we must include a `omp critical` section which runs the code block on each thread, one at a time; this prevents multiple threads trying to read and write to memory at the same time.

Reduction

The process detailed above (using `critical` sections) of combining multiple sub-components into a single components is known as reduction. We can write our own reduction code as above, but it can be implemented much more efficiently. Luckily, OpenMP provides a `reduction` directive which will efficiently reduce our code.

We add the `reduction` directive to the end of OpenMP's `parallel` directive, and the directive has the following form

```
reduction( operator : variable list)
```

For example, we may have thread-private calculations stored in `pvt_count` which we want to combine into a `total_count`. To do this we would use the directive

```
reduction(+ : total_count)
```

This tells the program that `total_count` will hold the global sum of a reduction.

We can write a simple C+ program to demonstrate reduction.

```
#include <iostream>

int main(void)
{
        int total_count = 0;

        #pragma omp parallel reduction( + : total_count)
        {
                int pvt_count = 0;

                #pragma omp for
                for(int i=0; i<100000; ++i)
                {
                        ++pvt_count;
                }

                total_count = total_count + pvt_count;
        }

        std::cout << "The total count is " << total_count << std::endl;

        return 0;
}
```

Note in the above that we still require the line `total_count = total_count + pvt_count`. We have simply told OpenMP that we will reduce the `pvt_counts` to a sum contained by `total_count`.